A More Efficient AES Threshold Implementation

Begül Bilgin[1,2], Benedikt Gierlichs[1], Svetla Nikova[1], Ventzislav Nikov[3], and Vincent Rijmen[1]

[1] KU Leuven, ESAT-COSIC and iMinds, Belgium
{name.surname}@esat.kuleuven.be
[2] University of Twente, EEMCS-DIES, The Netherlands
[3] NXP Semiconductors, Belgium venci.nikov@gmail.com

Abstract. Threshold Implementations provide provable security against first-order power analysis attacks for hardware and software implementations. Like masking, the approach relies on secret sharing but it differs in the implementation of logic functions. At Eurocrypt 2011 Moradi et al. published the to date most compact Threshold Implementation of AES-128 encryption. Their work shows that the number of required random bits may be an additional evaluation criterion, next to area and speed. We present a new Threshold Implementation of AES-128 encryption that is 18% smaller, 7.5% faster and that requires 8% less random bits than the implementation from Eurocrypt 2011. In addition, we provide results of a practical security evaluation based on real power traces in adversary-friendly conditions. They confirm the first-order attack resistance of our implementation and show good resistance against higher-order attacks.

Keywords: Threshold Implementation, First-order DPA, Glitches, Sharing, AES, S-box

1 Introduction

Embedded devices seem to be easily protected by modern ciphers in a black-box scenario. However, in the late 90s [10] the security of such devices has been shown to depend on the algorithm implementation. During the computation of an algorithm the device leaks information. Side channel attacks (SCA) are among the most relevant threats for the security of implementations of cryptographic algorithms. Certain countermeasures aim at introducing noise in the side channel, e.g. random delays, random order execution, dummy operations, etc., while masking conceals all sensitive intermediate values of a computation with random data and allows one to formally argue the security such a protection provides. Different masking schemes, like additive [8,9] and multiplicative [14], have been proposed in order to provide security against differential power analysis (DPA) attacks. However, it was shown [11,12,17] that masked hardware implementations can still be vulnerable to first-order DPA due to the presence of glitches. One can try to eliminate the security relevant glitches by carefully balancing

signal propagation delays, but this requires expertise, time, iterations of design and testing, and hence is expensive. As an alternative, new masking schemes have been developed that provide provable security even if glitches occur.

In 2006 Nikova et al. proposed such a scheme called Threshold Implementation (TI) [19]. It is based on secret-sharing and provably secure against first-order DPA [20]. In 2012 Prouff and Roche proposed a d^{th}-order masking scheme [24], based on Shamir's secret sharing, for which they claim security even against higher-order attacks. It is a general method that replaces every field multiplication by $4d^3$ field multiplications and $4d^3$ additions, using $2d^2$ bytes of randomness. In some cases this may prove too costly or inefficient. And [16] has shown that the multivariate leakages can be exploitable in univariate attacks.

Related Work. The Threshold Implementation technique is based on a specific type of multi-party computation and applies Boolean masking. Interesting properties of the technique are that it provides provable security against first-order side-channel attacks, that it requires few assumptions on the hardware leakage behavior, and that it allows to construct realistic-size circuits without intervention and design iterations. However, threshold implementations can still be broken by univariate mutual information analysis (MIA) [2,20] or univariate higher-order attacks [15].

It has been shown that all 3×3 and 4×4 have a TI sharing with 3, 4 or 5 shares [5]. The TI approach has been applied to only few entire algorithms: PRESENT [21], AES [18], KECCAK [3] and Fides [4]. In AES, the S-box is the by far most challenging part to share. Moradi et al. [18] have proposed a TI of this S-box that constantly uses 3 shares based on the tower field approach.

Contribution. We propose a more compact and faster Threshold Implementation of AES-128 encryption that requires less random bits compared to the one by Moradi et al. from EUROCRYPT 2011. For the S-box we use the tower field approach over $GF(2^4)$ and for each block in the S-box computation we adapt the number of shares. This reduces the area by 13% and the clock cycles by 40%. However, our main focus is to optimize not only the S-box but the whole cipher. Our implementation of AES is 18% smaller, 7.5% faster and requires 8% less random bits than the implementation from EUROCRYPT 2011. We investigate the uniformity problem and the need for re-masking in more detail. We prove that under certain circumstances, it is enough to re-mask only a fraction of the shares. We provide results of a practical security evaluation based on real power traces in adversary-friendly conditions. They confirm the theoretically guaranteed first-order attack resistance and show good security against higher-order attacks.

2 Threshold Implementation

TIs use sharings with the following properties: correctness, incompleteness and uniformity. The last property is often the most difficult to achieve. We propose

implementations where not every function satisfies the property of uniformity and use fresh randomness instead to do a re-masking. In this section, we recall the TI properties defined in [19] and describe how circuit complexity can be traded off for fresh random bits.

2.1 Notation and Definitions

We denote by upper-case characters stochastic variables, and by lower-case characters the values they can take, i.e. elements of a finite field. Let X, taking values in $\mathcal{F}^m$, denote the input of the (unshared) function f. A *masking* takes as inputs a value x and some auxiliary values (*random masks*), and outputs a vector $(x_1, \ldots, x_{s_x})$ such that the XOR-sum of the s_x shares equals x. For all values x with $\Pr(X = x) > 0$, let $\mathrm{Sh}(x)$ denote the set of valid share vectors $(x_1, \ldots, x_{s_x})$ for x:

$$\mathrm{Sh}(x) = \{(x_1, \ldots, x_{s_x}) \in \mathcal{F}^{ms_x} \mid x_1 + \cdots + x_{s_x} = x\} \ .$$

$\Pr((X_1, \ldots, X_{s_x}) = (x_1, \ldots, x_{s_x}) | X = x)$ denotes the probability that $(X_1, \ldots, X_{s_x}) = (x_1, \ldots, x_{s_x})$ when the input of the masking equals x, taken over all auxiliary inputs of the masking. Similarly, we denote the output of the unshared function by Y, taking values in $\mathcal{F}^n$, $(y_1, \ldots, y_{s_y})$ and $\mathrm{Sh}(y)$. Let F denote the vector function with input $(X_1, \ldots, X_{s_x})$ and output $(Y_1, \ldots, Y_{s_y})$; we will call it a *sharing*. TIs, like most other masking schemes, require that the masking is *uniform*, in the sense of the following definition.

Definition 1 (Uniform masking). *A masking is* uniform *if and only if for all x we have:*

$$\Pr((X_1, \ldots, X_{s_x}) = (x_1, \ldots, x_{s_x}) | X = x) = |\mathcal{F}|^{-m(s_x - 1)}$$

if $(x_1, \ldots, x_{s_x}) \in \mathrm{Sh}(x)$, else it is 0.

In words, we call a masking uniform if for each value x of the variable X, the corresponding vectors with masked values occur with the same probability.

Threshold implementations use sharings that satisfy the following properties. Firstly, the sharing F of f needs to be *correct*:

$$\forall y \in \mathcal{F}^n, \forall (x_1, \ldots, x_{s_x}) \in \mathrm{Sh}(x), \forall (y_1, \ldots, y_{s_y}) \in \mathrm{Sh}(y) : \\ F(x_1, \ldots, x_{s_x}) = (y_1, \ldots, y_{s_y}) \Leftrightarrow f(x) = y.$$

Secondly, the sharing needs to be *incomplete*: every component function of F that outputs Y_i should be independent of at least one share X_i. The third property is *uniformity of the sharing* [19]. Although the main point of this section is that also sharings which do not satisfy the third property can be used in threshold implementations, we provide the definition already now.

Definition 2 (Uniform sharing). *The sharing F of f is* uniform *if and only if*

$$\forall y \in \mathcal{F}^n, \forall (y_1, \ldots, y_{s_y}) \in \mathrm{Sh}(y), \forall x \in \mathcal{F}^m \text{ with } f(x) = y : \\ \left| \left\{ (x_1, \ldots, x_{s_x}) \in \mathrm{Sh}(x) | F(x_1, \ldots, x_{s_x}) = (y_1, \ldots, y_{s_y}) \right\} \right| = \frac{|\mathcal{F}|^{m(s_x - 1)}}{|\mathcal{F}|^{n(s_y - 1)}} \ .$$

If $s_x = s_y$ and $m = n$, this simplifies to:

$$\forall y \in \mathcal{F}^n,\ \forall (y_1, y_2, \ldots, y_{s_y}) \in \mathrm{Sh}(y) \forall x \in \mathcal{F}^n \text{ with } f(x) = y : \\ \left|\left\{(x_1, x_2, \ldots, x_{s_x}) \in \mathrm{Sh}(x) | F(x_1, x_2, \ldots, x_{s_x}) = (y_1, y_2, \ldots, y_{s_y})\right\}\right| = 1 \ .$$

It follows that in this case a uniform sharing F is invertible if and only if f is invertible.

2.2 Security from Correctness and Incompleteness

The security of threshold implementations against first-order side-channel attacks follows from two intuitively easy steps. If the masking is uniform and the sharing F is incomplete, then

1. any single component function of F does not get the information to determine the value of X (*it does not know* x), hence cannot leak any information on X, and
2. the expected value (average) of any leakage signal of an implementation of the sharing F, be it instantaneous or summed over an arbitrary period of time, is constant.

Note that the only assumption on the physical behavior of the hardware or software implementation of F that is needed for this reasoning, is that it should be possible to implement the component functions in such a way that they are each independent of one share X_i. In other words, the cross-talk between implementations of different components should be negligible.

2.3 Uniformity for the Cascaded and Parallel Functions

If the threshold implementation technique is used to protect cascaded functions, then extra measures need to be taken, such that the input for the next non-linear operation is again a uniform masking. A similar situation occurs when the threshold implementation technique is used to protect several functional blocks acting in parallel on (partially) dependent inputs. This occurs for example in implementations of the AES S-box using the tower field approach. If no special care is taken, then "local uniformity" of the distributions of the inputs of the individual blocks will not lead to "global uniformity", i.e. for the joint distributions of the inputs of all blocks. For example, let g and h be two functions acting on the same input X. Then, even if G and H are uniform sharings, producing uniform $Y = G(X)$ and $Y' = H(X)$, this does not imply that (Y, Y') is uniform. If each of the parallel blocks satisfies the properties of correctness and incompleteness, there will be no leakage of signals within the parallel blocks. However, the lack of uniformity in the joint distribution of the masking of the outputs can lead to information leakage if the outputs are combined as inputs to a next function.

We can take different types of actions to remedy this problem. The *first approach* is to require uniformity of the sharing F (Definition 2). We can show

that if the sharing is uniform and the masking of its input is uniform, then also the masking of its output is uniform.

Theorem 1. *If the masking of X is uniform and the sharing F is uniform, then the masking of $Y = f(X)$, defined by $(y_1, \ldots, y_{s_y}) = F(x_1, \ldots, x_{s_x})$, is uniform.*

The proof is omitted here to save space. Practice shows that adding the uniformity requirement to a sharing tends to blow up the mathematical complexity of the sharing, as well as the cost of its implementation. In some applications, it might be better to consider a *second approach*: re-masking as for example done by Moradi et al. [18]. Indeed, by adding new random masks to the shares, we can make the distribution uniform.

2.4 Reducing the Randomness Used in a Re-masking Step

The following theorem allows to reduce the amount of random bits used by re-masking steps of threshold implementations: under certain circumstances, only a fraction of the shares needs to be re-masked.

Theorem 2. *Let $(X_1, \ldots, X_s)$ be a sharing of a variable $X \in \mathcal{F}^m$, where $\Pr(X_1 = x_1, \ldots, X_t = x_t) = |\mathcal{F}|^{-tm}, \forall (x_1, \ldots, x_t)$ for some t with $1 \leq t \leq s$. Then the sharing $(Y_1, \ldots, Y_s)$, defined by $Y_i = X_i$ for $1 \leq i \leq t$ and $Y_i = X_i + R_i$ for $t < i \leq s$, is a* uniform *sharing for X, i.e.: $\Pr(Y_1 = y_1, \ldots, Y_s = y_s | X = y_1 + \cdots y_s) = |\mathcal{F}|^{(1-s)m}$, provided that the R_i, $i = t+1, \ldots, s-1$ are independently and uniformly distributed random variables and that $R_s = -(R_{t+1} + \cdots + R_{s-1})$.*

Proof. We give here a sketch of the proof. We have:

$$\begin{aligned} &\Pr(Y_1 = y_1, \ldots, Y_s = y_s | X = y_1 + \cdots y_s) \\ = &\Pr(Y_1 = y_1, \ldots, Y_t = y_t | X = y_1 + \cdots y_s) \\ &\cdot \Pr(Y_{t+1} = y_{t+1}, \ldots, Y_s = y_s | X = y_1 + \cdots y_s, Y_1 = y_1, \ldots, Y_t = y_t) . \end{aligned} \tag{1}$$

Since $Y_i = X_i$ for $1 \leq i \leq t$, the first factor equals $|\mathcal{F}|^{-tm}$. For the second factor we recall the definition of Y_{t+1} to obtain that:

$$\Pr(Y_{t+1} = y_{t+1}) = \sum_{x_{t+1}} \Pr(X_{t+1} = x_{t+1}) \underbrace{\Pr(R_{t+1} = y_{t+1} - x_{t+1})}_{|\mathcal{F}|^{-m}} .$$

The same holds for $Y_{t+2}, \ldots, Y_{s-1}$ and since the R_i have independent distributions, we can equate the second factor of (1) to:

$$|\mathcal{F}|^{(1-s-t)m} \sum_{x_{t+1}, \ldots, x_{s-1}} \Pr\left(X_{t+1} = x_{t+1}, \ldots, X_{s-1} = x_{s-1}, Y_s = y_s \middle| X = y_1 + \cdots + y_s, X_1 = x_1, \ldots, X_t = x_t\right) .$$

Recalling the definition of Y_s completes the proof. □

Note that generating the extra randomness required by the re-masking approach may become a bigger challenge in some cases than the blow-up in gate count caused by the uniform sharing approach.

Conclusion. Assume that we have an input that is uniformly masked. Section 2.2 explains that single circuits are secure against first-order side-channel attacks, if they satisfy the incompleteness property. Section 2.3 explains that for cascaded circuits we need to ensure that the inputs of all circuits are uniformly masked. This can be done either by using uniform sharings (Def. 2) or by re-masking. The point that we want to stress here, however is that we do not need to do both: an implementation that uses re-masking, does not need uniform sharings in order to resist first-order attacks.

By relinquishing the uniformity requirement, it is often possible to reduce the number of shares and the size of the implementation. This will be used in the next section in order to reduce the number of shares in the subblocks of the AES S-box and improve on the implementation of [18].

3 Implementation

In this section, we will discuss the new TI of AES in detail. We will first describe the general data flow of our implementation. Then we will introduce a new approach to apply the TI to the S-box of AES which is the only non-linear layer of the block cipher. We used ModelSim to verify the functionality of the proposed design and Synopsys Design Vision D-201-.03-SP4 with Faraday Standard Cell Library FSA0A_C_Generic_Core, which is based on UMC 0.18μm GenericII Logic Process with 1.8V voltage, for synthesis. We will conclude this section by providing the performance of our design together with the comparison with the previous work in [18]. We should note that the work in [18] uses a similar standard cell library based on UMC 0.18μm logic process with 1.8V voltage.

3.1 General Data Flow

Our main goal for this implementation is to minimize the area and randomness overhead caused by the sharing. To achieve this, we use a serial implementation as proposed in [18] which requires only one S-box instance and loads the plaintext and key byte-wise in row-wise order. Moreover, we adapt the number of shares used in each operation in the block cipher. That is, we use two shares which is the minimum number of shares possible for the affine operations such as MixColumns or Key XOR and increase or decrease the number of shares when required for the non-linear layer. This can also be seen in Fig. 1, as the key and the state registers are 256 bits implying the two shares. With this approach we already decrease a significant part of the register cost since one bit register costs 5.33 GE in our library.

The TI of the S-box, for which the details will be given in the following section, requires four input shares and 20 bits of randomness and outputs three shares. Therefore our initial sharing for the plaintext is also with four shares. However, it is enough to initialize the sharing of the key with two shares. More details about the key scheduling will be given later in this section. The two shares of the key are XORed with two of the plaintext shares before the S-box

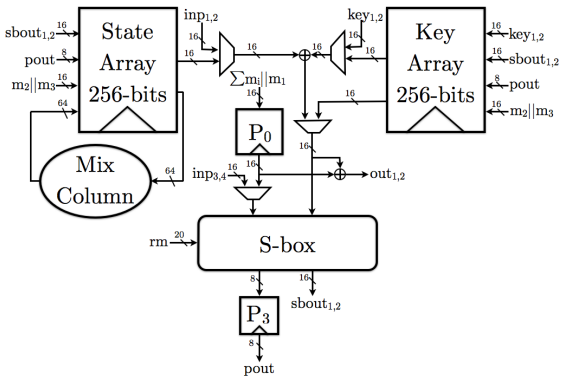
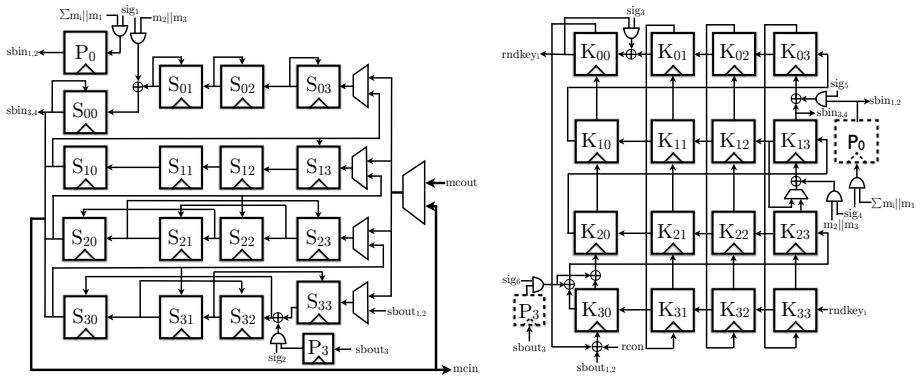


Fig. 1: Architecture of the serialized TI of AES-128 .

operation. After three clock cycles the first output share of the S-box operation is written to the register P_3 whereas the remaining two shares are written to the state register S_{33}. The data in P_3 is XORed with the second share of the S-box output, in the state register S_{33}, after one clock cycle to be able to continue with two shares for the linear operations. In the following AES rounds, we increase the number of shares from two to four by using 24 bits of randomness one clock cycle before the S-box operation. We store the additional two shares in P_0 to achieve the non-completeness property in the following combinational logic. The registers P_0 and P_3 are used both for the round transformations and the key scheduling.

State Array (Fig. 2a) The state array consists of sixteen 16-bit registers each corresponding to the two shares of a byte in the state. From the first to the sixteenth clock cycle, the four input shares (first round) or the shares in the registers S_{00} and P_0 (later rounds) are sent to the S-box module. The corresponding three output shares are written to the registers S_{33} and P_3. The signal sig_2 is active from the fourth to the nineteenth clock cycle to reduce the number of shares from three to two in the state such that one of the shares in S_{33} is XORed with P_3 and the other share stays the same. The state is shifted to the left horizontally from the third to the eighteenth clock cycle. The Shift Rows operation is also completed in the nineteenth clock cycle with an irregular horizontal shift. In the next four clock-cycles, the data in the registers S_{00}, S_{10}, S_{20} and S_{30} are sent to MixColumns operation, the rest of the registers are shifted to the left horizontally and the output of the MixColumns operation is written to the registers S_{03}, S_{13}, S_{23} and S_{33}. The MixColumns operation is implemented column-wise as in [18] and with two shares working in parallel. The registers except S_{10}, S_{11} and S_{12} are implemented as scan flip-flops (SFF) that are D-flip-flops (DFF) combined with 2-to-1 MUXes and can operate with two inputs to reduce the area since a single 2-to-1 MUX costs 3.33 GE in our library whereas one bit SFF costs 6.33 GE. One round of AES takes 23 clock



(a) State array with ShiftRows (b) Key array

Fig. 2: Architecture of the registers where S_i, K_i and P_0 hold two shares and P_3 holds one share. The registers P_0 and P_3 are shared by the state and the key array. The XOR of the value in P_3 and S_{33} (resp. K_{30}) is on one share of the value in register S_{33} (resp. K_{30}) whereas all the other combinational operations are on two shares.

cycles. The signal sig_1 is active for sixteen clock cycles, starting from the last clock-cycle of each round, to increase the number of shares from two to four.

Key Array (Fig. 2b) Similar to the state array, the key array also consists of sixteen 16-bit registers implemented as SFFs each corresponding to the two shares of a byte in the key schedule. The round key is inserted from the register K_{33} in the first sixteen clock cycles of each round. For the next three clock cycles, the registers except K_{03}, K_{13}, K_{23} and K_{33} are not clocked. The registers K_{03}, K_{23} and K_{33} are also not clocked in the seventeenth clock cycle. In that clock cycle, we increase the number of shares in the register K_{13}. In the following three clock cycles this re-sharing is done during the vertical shift from the register K_{23} to K_{13}. Hence the re-sharing signal sig_4 is active from the seventeenth to the twentieth clock cycle. Signal sig_5 is active from the eighteenth to the twenty-first clock cycle to reduce the number of shares back to two. The registers K_{03}, K_{13}, K_{23} and K_{33} are not clocked in the remaining two clock cycles of each round. We choose this way of irregular clocking to avoid using extra MUXes in our design. Two shares of the S-box output are XORed to the data in K_{00} in the last four clock cycles of each round. In the twentieth clock cycle the round counter *rcon* is additionally XORed to one of these shares. The number of shares is reduced back to two by XORing the share in P_3 to one of the shares in K_{30}. Signal sig_3 is active in the first sixteen clock cycles except the fourth, eighth, twelfth and sixteenth clock cycles. The roundkey is taken from the register K_{00} to be XORed with the corresponding plaintext before going to the S-box operation.

3.2 TI of the AES S-box

The S-box (Fig. 3) is instantiated only once to be used by both the key schedule and the state update. In the first sixteen clock cycles, it gets its inputs from the state. The input is taken from the key array in clock cycles eighteen to twenty-one.

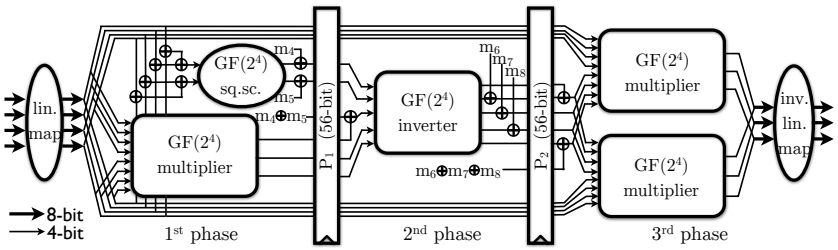


Fig. 3: The Sbox of our implementation.

The S-box implementation in [18] uses the tower field approach up to $GF(2^2)$ for a smaller implementation. Therefore, the only non-linear operation is $GF(2^2)$ multiplication which must be followed by registers to avoid first order leakages.

We also chose to use the tower field approach, however, we decided to go to $GF(2^4)$ instead of $GF(2^2)$. With this approach, the $GF(2^4)$ inverter can be seen as a four bit permutation and the $GF(2^4)$ multiplier as a four bit multiplication both of which are well studied in [6]. Therefore, we can find uniform TIs for these non-linear blocks individually which implies using less fresh random bits during the combination of these uniformly implemented pieces. Moreover, with this approach the S-box calculation takes three clock cycles instead of five.

The algebraic normal form of the multiplier in $GF(2^4)$ is given in Appendix A.1. This multiplication can be shared uniformly as in Appendix A.3 with four input and three output shares. The required area is 625 GE without any optimization.

The $GF(2^4)$ inverter, on the other hand, can be represented with the formula in Appendix A.2. To have a uniform sharing for this function, which belongs to class $\mathcal{C}^4_{282}$ [5], we consider two options. Either using four shares which is the minimum number of shares necessary for a uniform implementation in that class and decomposing the function into three uniform sub-functions as $Inv(x) = F(G(H(x)))$, or using five shares without any decomposition. Our experiments show that both versions have similar area requirements but a different number of clock cycles. To reduce the number of cycles, we chose the version with five shares, with the formula in Appendix A.4, which requires 618 GE. The sharing for this module is found by using the method described in [20] which is slightly different from the direct sharing [5]. We chose this formula since it can be implemented with less logic gates in hardware compared to the direct sharing.

Even though it is enough to use only two shares for linear operations, we sometimes chose to work on more than two shares to avoid the need for extra

random bits. The linear map of the tower-field S-box operates on four shares since the multiplication needs four input shares. The inverter requires five input shares and the multiplication outputs only three shares, therefore we use two shares for the square scalar to have five shares in the beginning of the second phase. We use three shares for the inverse linear map of the tower-field S-box since the multiplication outputs three shares.

Combining the sub-blocks. During this process we face two challenges. One is to keep the uniformity in the pipeline registers as the sub-blocks are combined. That is a challenge Moradi et al. also faced and solved with re-masking. We also apply re-masking in the second phase where we combine the 2 output shares of the square scaler and the 3 output shares of the multiplier to 5 shares. We must note that this combination also acts as the XOR of the output of the square scaler and multiplier in the unshared case. By theorem 2, it is enough to re-mask only the output shares from one function to achieve uniformity. We choose to re-mask the output of the square scaler since it operates on less shares hence requires less random bits. The correction mask, i.e. the XOR of the masks, is XORed to one of the output shares of the multiplier to achieve correctness and non-completeness.

The second challenge is to keep the uniformity as we increase or decrease the number of shares. This is achieved by introducing new masks before the S-box operation to increase from two to four shares and at the end of the second phase to decrease from five to four shares. The output of the third phase is not uniform when the three shares are considered together. However, we verified by simulation that each share individually is uniform which implies that there is no first-order leakage in the following register. We combine the first two shares with an XOR and keep the third share as it is to go back to two shares. We also verified that, when we decrease the number of shares to two, the output shares are uniform.

We always keep the XOR of the masks in the pipeline registers and complete the re-masking in the next clock cycle as in [18]. Overall, we need 44 fresh random bits per S-box operation which is less than what was required in [18].

3.3 Performance

Like other countermeasures TIs require extra area and randomness. In this work we minimize these needs for a more efficient implementation. In Table 1, we show the area, randomness and timing requirements of our implementation and compare them with [18]. The area cost for the state and the key arrays include the ANDs and XORs that are in Fig. 2. An expected observation is that the cost of the state and key array together with the MixColumns is reduced by one third compared to [18] since we use two shares instead of three. The area cost of the S-box is the sum of the combinational logic in three phases and the registers required. For the three phases, we use four linear maps (each 42 GE), two square scalers (each 9 GE), three multipliers (each 625 GE), one inverter (618 GE), three inverse linear maps (each 33 GE) and some additional XORs

Table 1: Synthesis results for different versions of AES TI.

	State Array	Key Array	S-box	MixCol Col	Contr.[1]	Key XOR	MUX	Other	Total	cycles	rand bits[2]
[18]	2529	2526	4244	1120	166	64	376	89	11114/11031[3]	266	48
This paper	1698	1890	3708	770	221	48	746	21	9102	246	44
This paper[3]	1698	1890	3003	544	221	48	746	21	8171	246	44

[1] including round constant [2] per S-box [3] *compile_ultra*

for re-masking. The registers P_0 and P_3 are also counted in the cost of the S-box together with the pipelining registers P_1 and P_2.

In this implementation, the S-box occupies 40% of the total area. When compared to the previous implementation by Moradi et al., the S-box is 13% smaller and the overall area is 18% smaller. Moreover it is faster and requires less randomness. The numbers provided in Table 1 are taken from the Synopsys tool with *compile* command. We use these numbers for a fair quantitative comparison. On the other hand, it is also possible to compile each function that is provided in Appendix A.3 and A.4 *individually* with the *compile_ultra* command to let the tool optimize these functions and use the generated optimized descriptions of these functions. This reduces the cost of TI of AES to 8171 GE. However, the results for *compile_ultra* mainly reflect how good the tools are at optimizing and a comparison may not be fair.

4 Power Analysis

To evaluate the security of our design in practice we implement it on a SASEBO-G board [1] using Xilinx ISE version 10.1. We use the "keep hierarchy" constraint to prevent the tools from optimizing over module boundaries (see the last paragraph of Sect. 2.2). The board features two Xilinx Virtex-II Pro FPGA devices: we implement the TI AES and a PRNG on the crypto FPGA (xc2vp7) while the control FPGA (xc2vp30) handles I/O with the measurement PC and other equipment. The PRNG that generates all random bits is implemented as AES-128 in CTR mode.

We measure the power consumption of the crypto FPGA during the first 1.5 rounds of TI AES as the voltage drop over a 1Ω resistor in the FPGA core GND line. The output of the passive probe is sampled with a Tektronix DPO 7254C digital oscilloscope at 1GS/s sampling rate.

Methodology. We define two main goals for our practical evaluation. First, we want to verify our implementation's resistance against first-order attacks. But in practice adversaries are of course not restricted to applying such attacks. Therefore, our second goal is to assess the level of security our implementation provides against other, e.g. higher-order, power analysis attacks.

Since there is no single, all-embracing test to evaluate the security of an implementation, we follow the approach of [18] and test its resistance against state-of-the-art attacks. We narrow the evaluation to univariate attacks because our implementation processes all shares of a value in parallel. Estimating the

information-theoretic metric by Standaert et al. [25] is out of reach. It would require estimation of up to 2^{56} Gaussian templates.

We make several choices that are in favor of an adversary and make attacks easier. First, to minimize algorithmic noise the PRNG and the TI AES do not operate in parallel, i.e. the PRNG generates and stores a sufficient number of random bits before each TI AES operation. In practice, running them in parallel will increase the level of noise and thus the number of measurements needed for an attack to succeed. Second, we provide the crypto FPGA with a stable 3MHz clock frequency to ensure that the traces are well aligned and the power peaks of adjacent clock cycles do not overlap (this would also help to assign a possibly identified leak to a specific clock cycle). In practice, clocking the device at a faster or unstable clock will make attacks harder. Note that the "combining effect" of the measurement setup or a faster clock described in [16] does not apply to our situation. In our implementation all shares are processed and leak at the same time, in contrast to the implementation analyzed in [16] where all shares are processed and leak separated in time. Hence we expect the effect to not ease an attack. Third, we let the adversary know the implementation. Specifically, if the PRNG was switched off the adversary would be able to correctly compute bit values and bit flips under the correct key hypothesis. In practice, obscurity is often used as an additional layer of security. Fourth, we use synchronous sampling [13] to avoid clock drift and achieve the best possible alignment. In practice, secure devices use an internal (and unstable) clock source which prevents synchronous sampling and increases the number of measurements needed for an attack to succeed.

PRNG switched off. To confirm that our setup works correctly and to get some reference values we first attack the implementation with the PRNG switched off. We expect that the implementation can be broken with many first-order attacks. As example, Fig. 4 shows the result of a correlation DPA attack [7] that uses the Hamming distance of two consecutive S-box outputs as power model. The attacks require $2 \cdot 2^8$ key hypotheses. To reduce the computational complexity we let the adversary know one key byte and aim to recover the second one.

Since the adversary knows the implementation, he can choose to compute the Hamming distance over three 8-bit registers (S_{33} and P3; output of the S-box in three shares), two 8-bit registers (S_{32}; one cycle later; two shares) or ignore the details and compute the distance over a single 8-bit register as if it was a plain implementation. The results for all three options are identical. This is a property of our implementation that vanishes when the PRNG is switched on. Only a few hundred traces are required to recover the key with one of these attacks. It is worth noticing that the highest correlation peak does not occur at the S-box output registers, but three resp. two clock cycles later when the bit-flips occur in register S_{30}. This register drives the MixColumns logic and therefore has a much greater fanout.

Fig. 5 shows the result of a correlation collision attack [17] that targets combinational logic. The attack computes two sets of mean traces for the values of

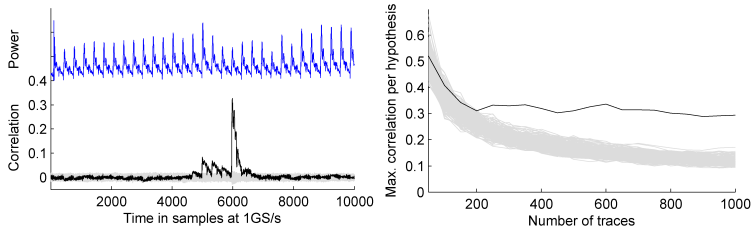


Fig. 4: Results of DPA attacks using HD model over 3/2/1 registers with PRNG off; left: correlation traces for all key hypotheses computed using 50 000 power traces, correct hypothesis in black, and a scaled power trace; right: max. correlation coefficient per key hypothesis (from the overall time span) over number of traces used.

two processed plaintext bytes and shifts the mean traces in the time domain to align them. It aims to recover the linear difference between the two key bytes involved. To do so, it permutes one set of mean traces according to a hypothesis on the linear difference and then correlates both sets of mean traces. The result shows that this attack is successful with a few thousand measurements.

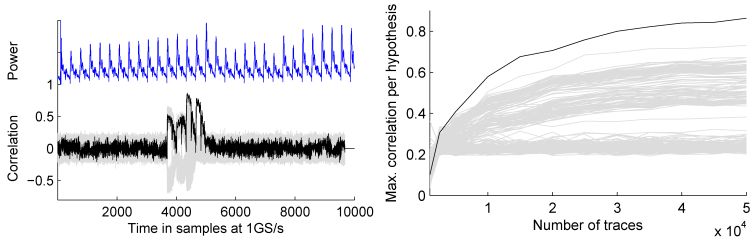


Fig. 5: Result of a correlation collision attack with PRNG off; left: correlation traces for all hypotheses on the linear difference computed using 50 000 power traces, correct hypothesis in black, and a scaled power trace; right: max. correlation coefficient per hypothesis on the linear difference (from the overall time span) over number of traces used.

PRNG switched on. Next we repeat the evaluation with the PRNG switched on, i.e. the TI AES uses unknown and unpredictable random bits. However, for the DPA attacks using the Hamming distance over two or three registers as power model we again suppose these bits were zero. Fig. 6 shows the results of the first-order attacks against the protected implementation using 10 million measurements. The results show that the attacks fail.

We proceed with higher-order attacks to assess the level of security our implementation provides. For our second-order DPA attacks we use the same power models as before but center and then square the traces (for each time sample)

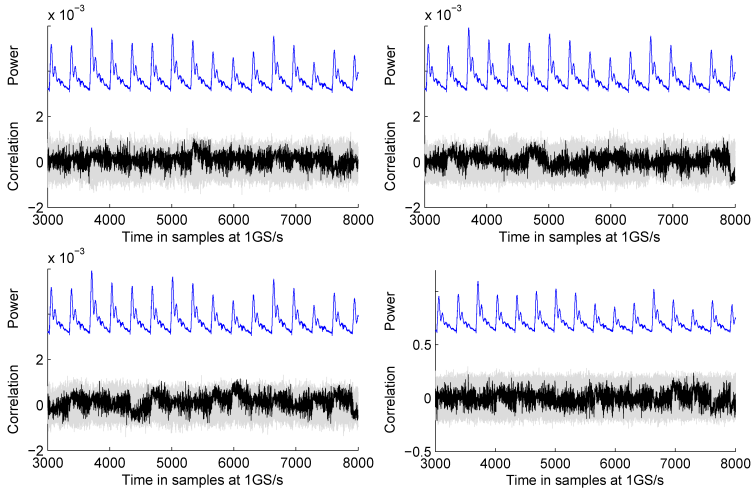


Fig. 6: Results of first-order DPA and correlation collision attacks with PRNG on computed using 10 million traces; top, left: HD over 1 register; top, right: HD over 2 registers; bottom, left: HD over 3 registers; bottom, right: correlation collision.

before correlating [8,23,26]. Second-order correlation collision attacks work as above with mean traces replaced by variance traces [15].

Fig. 7 (top) shows the results of the second-order DPA attack that uses the Hamming distance in a single register as power model (as if it was a plain implementation). The attack requires about 600 000 traces to succeed. We note that the highest correlation peak occurs again when the bitflips happen in register S_{30}, cf. Fig. 4. Second-order DPA attacks using the Hamming distance over two resp. three registers as power model failed to recover the key.

Fig. 7 (bottom) shows the results of the second-order correlation collision attack. The attack requires about 3.5 million traces to succeed. A third-order correlation collision attack works as above with mean traces replaced by skewness traces [15]. This attack fails using 10 million measurements.

Discussion. The first goal of our evaluation is to verify our implementation's resistance against first-order attacks. But this goal is always limited by the number of measurements at hand. It is simply not possible to demonstrate resistance against attacks with an infinite number of traces. We have shown that our implementation resists state-of-the-art first-order attacks with 10 million traces in conditions that are strongly in favor of the adversary (no algorithmic noise from the PRNG, knowledge of the implementation, slow and stable clock, best possible alignment). Given the theoretical foundations of TI and the correctness of our implementation, we are convinced that our implementation resists first-order

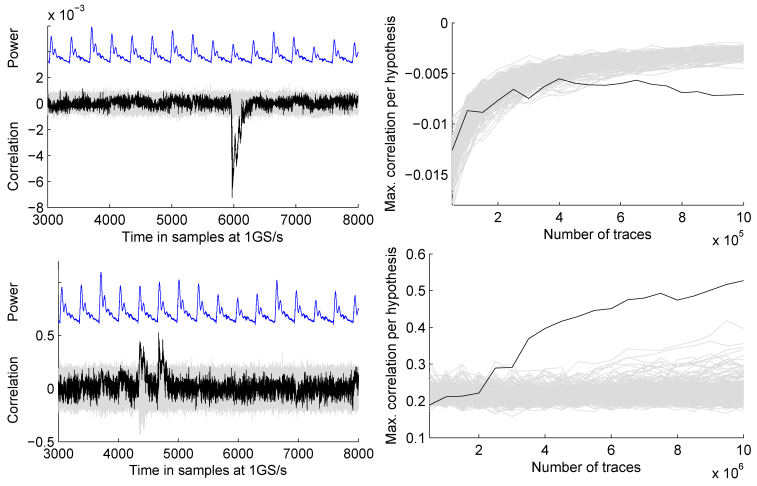


Fig. 7: Results of second-order DPA (top) and correlation collision (bottom) attacks with PRNG on computed using 10 million traces; right: min./max. correlation coefficient per hypothesis (from the overall time span) over number of traces used.

attacks with any number of measurements, but we have no way to demonstrate that.

The second goal of our evaluation is to assess the level of security our implementation provides against other attacks. In the same adversary-friendly conditions, the most trace-efficient second-order attack in our evaluation requires about 600 000 traces. Recall that our evaluation focuses on univariate attacks, so that the computational overhead is limited to estimating second-order moments and does not involve the notoriously more costly search over pairs of points in time. However, regarding second-order attacks it is well known that the number of traces required for an attack to succeed scales quadratically in the noise standard deviation [8,22]. Therefore, second-order attacks against our implementation in less favorable and more realistic, i.e. much more noisy, conditions (algorithmic noise from the PRNG, no knowledge of the implementation, faster and unstable clock, worse alignment) will require many more traces.

It is tempting to compare the results of our evaluation to the results of the evaluation in [18]. However, not only the implementations but also the measurement platforms and the conditions differ, so that any difference must not be credited to an implementation alone. Already the numbers of traces required for attacks against the implementations with PRNG switched off differ by roughly two orders of magnitude. In addition, the analysis in [18] is limited to four clock cycles during the S-box computation.

5 Acknowledgements

This work has been supported in part by the Research Council of KU Leuven (OT/13/071), by the FWO (G.0550.12), by the Hercules foundation and by GOA (tense). B. Bilgin was partially supported by the FWO project G0B4213N, V. Nikov was supported by the European Commission (FP7) within the Tamper Resistant Sensor Node (TAMPRES) project with contract number 258754 and Benedikt Gierlichs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

References

1. AIST. Side-channel Attack Standard Evaluation BOard. `http://staff.aist.go.jp/akashi.satoh/SASEBO/en/`.
2. L. Batina, B. Gierlichs, E. Prouff, M. Rivain, F.-X. Standaert, and N. Veyrat-Charvillon. Mutual Information Analysis: a Comprehensive Study. *J. Cryptol.*, 24(2):269–291, April 2011.
3. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Building power analysis resistant implementations of KECCAK. Second SHA-3 candidate conference, August 2010.
4. B. Bilgin, A. Bogdanov, M. Kneževic, F. Mendel, and Q. Wang. Fides: Lightweight authenticated cipher with side-channel resistance for constrained hardware. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 142–158. Springer Berlin Heidelberg, 2013.
5. B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, and G. Stütz. Threshold implementations of all 3×3 and 4×4 S-boxes. In *CHES*, volume 7428 of *LNCS*, pages 76–91. Springer, 2012.
6. B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, and G. Stütz. Threshold implementations of all 3×3 and 4×4 S-boxes. Cryptology ePrint Archive, Report 2012/300, 2012. `http://eprint.iacr.org/`.
7. E. Brier, C. Clavier, and F. Olivier. Correlation power analysis with a leakage model. In *CHES*, volume 3156 of *LNCS*, pages 16–29. Springer, 2004.
8. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.
9. L. Goubin and J. Patarin. DES and differential power analysis the "duplication" method. In *CHES*, volume 1717 of *LNCS*, pages 158–172. Springer, 1999.
10. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
11. S. Mangard, T. Popp, and B. M. Gammel. Side-channel leakage of masked CMOS gates. In *CT-RSA*, volume 3376 of *LNCS*, pages 351–365. Springer, 2005.
12. S. Mangard, N. Pramstaller, and E. Oswald. Successfully attacking masked AES hardware implementations. In *CHES*, volume 3659 of *LNCS*, pages 157–171. Springer, 2005.
13. T. S. Messerges. *Power analysis attacks and countermeasures on cryptographic algorithms*. PhD thesis, University of Illinois at Chicago, 2000.
14. T. S. Messerges. Securing the AES finalists against power analysis attacks. In Bruce Schneier, editor, *FSE*, volume 1978 of *LNCS*, pages 150–164. Springer, 2000.

15. A. Moradi. Statistical tools flavor side-channel collision attacks. In D. Pointcheval and T. Johansson, editors, *EUROCRYPT*, volume 7237 of *LNCS*, pages 428–445. Springer, 2012.
16. A. Moradi and O. Mischke. On the simplicity of converting leakages from multivariate to univariate - (case study of a glitch-resistant masking scheme). In G. Bertoni and J.-S. Coron, editors, *CHES*, volume 8086 of *LNCS*, pages 1–20. Springer, 2013.
17. A. Moradi, O. Mischke, and T. Eisenbarth. Correlation-enhanced power analysis collision attack. In *CHES*, volume 6225 of *LNCS*, pages 125–139. Springer, 2010.
18. A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. Pushing the limits: A very compact and a threshold implementation of AES. In *EUROCRYPT*, volume 6632 of *LNCS*, pages 69–88. Springer, 2011.
19. S. Nikova, C. Rechberger, and V. Rijmen. Threshold implementations against side-channel attacks and glitches. In *ICICS*, volume 4307 of *LNCS*, pages 529–545. Springer, 2006.
20. S. Nikova, V. Rijmen, and M. Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *J. Cryptology*, 24(2):292–321, 2011.
21. A. Poschmann, A. Moradi, K. Khoo, C.-W. Lim, H. Wang, and S. Ling. Side-channel resistant crypto for less than 2300 GE. *J. Cryptology*, 24(2):322–345, 2011.
22. E. Prouff and M. Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *LNCS*, pages 142–159. Springer, 2013.
23. E. Prouff, M. Rivain, and R. Bevan. Statistical analysis of second order differential power analysis. *IEEE Trans. Computers*, 58(6):799–811, 2009.
24. E. Prouff and T. Roche. Higher-order glitches free implementation of the AES using secure multi-party computation protocols. In *CHES*, volume 6917 of *LNCS*, pages 63–78. Springer, 2011.
25. F.-X. Standaert, T. Malkin, and M. Yung. A unified framework for the analysis of side-channel key recovery attacks. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *LNCS*, pages 443–461. Springer, 2009.
26. J. Waddle and D. Wagner. Towards efficient second-order power analysis. In M. Joye and J.-J. Quisquater, editors, *CHES*, volume 3156 of *LNCS*, pages 1–15. Springer.

A Equations

A.1 Multiplier in $GF(2^4)$

$(f_1, f_2, f_3, f_4) = (x_1, x_2, x_3, x_4) \times (x_5, x_6, x_7, x_8)$

$$f_1 = x_1x_5 \oplus x_3x_5 \oplus x_4x_5 \oplus x_2x_6 \oplus x_3x_6 \oplus x_1x_7 \oplus x_2x_7 \oplus x_3x_7 \oplus x_4x_7 \oplus x_1x_8 \oplus x_3x_8$$
$$f_2 = x_2x_5 \oplus x_3x_5 \oplus x_1x_6 \oplus x_2x_6 \oplus x_4x_6 \oplus x_1x_7 \oplus x_3x_7 \oplus x_2x_8 \oplus x_4x_8$$
$$f_3 = x_1x_5 \oplus x_2x_5 \oplus x_3x_5 \oplus x_4x_5 \oplus x_1x_6 \oplus x_3x_6 \oplus x_1x_7 \oplus x_2x_7 \oplus x_3x_7 \oplus x_1x_8 \oplus x_4x_8$$
$$f_4 = x_1x_5 \oplus x_3x_5 \oplus x_2x_6 \oplus x_4x_6 \oplus x_1x_7 \oplus x_4x_7 \oplus x_2x_8 \oplus x_3x_8 \oplus x_4x_8$$

A.2 Inverter in $GF(2^4)$

$(f_1, f_2, f_3, f_4) = Inv(x_1, x_2, x_3, x_4)$

$$
\begin{aligned}
f_1 &= x_3 \oplus x_4 \oplus x_1x_3 \oplus x_2x_3 \oplus x_2x_3x_4 \\
f_2 &= x_4 \oplus x_1x_3 \oplus x_2x_3 \oplus x_2x_4 \oplus x_1x_3x_4 \\
f_3 &= x_1 \oplus x_2 \oplus x_1x_3 \oplus x_1x_4 \oplus x_2x_2x_4 \\
f_4 &= x_2 \oplus x_1x_3 \oplus x_1x_4 \oplus x_2x_4 \oplus x_1x_2x_3
\end{aligned}
$$

A.3 Sharing Multiplier in $GF(2^4)$ with 4 Input 3 Output Shares

$$
\begin{aligned}
f &= xy, \ \text{where} \\
f &= f_1 \oplus f_2 \oplus f_3 \\
x &= x_1 \oplus x_2 \oplus x_3 \oplus x_4 \\
y &= y_1 \oplus y_2 \oplus y_3 \oplus y_4
\end{aligned}
$$

$$
\begin{aligned}
f_1 &= (x_2 \oplus x_3 \oplus x_4)(y_2 \oplus y_3) \oplus y_4 \\
f_2 &= ((x_1 \oplus x_3)(y_1 \oplus y_4)) \oplus x_1y_3 \oplus x_4 \\
f_3 &= ((x_2 \oplus x_4)(y_1 \oplus y_4)) \oplus x_1y_2 \oplus x_4 \oplus y_4
\end{aligned}
$$

A.4 Sharing Inverter in $GF(2^4)$ with 5 Input 5 Output Shares

$$
\begin{aligned}
f &= xyz \oplus xy \oplus z, \ \text{where} \\
f &= f_1 \oplus f_2 \oplus f_3 \oplus f_4 \\
x &= x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \\
y &= y_1 \oplus y_2 \oplus y_3 \oplus y_4 \oplus y_5 \\
z &= z_1 \oplus z_2 \oplus z_3 \oplus z_4 \oplus z_5
\end{aligned}
$$

$$
\begin{aligned}
f_1 &= ((x_2 \oplus x_3 \oplus x_4 \oplus x_5)(y_2 \oplus y_3 \oplus y_4 \oplus y_5)(z_2 \oplus z_3 \oplus z_4 \oplus z_5)) \\
&\quad \oplus ((x_2 \oplus x_3 \oplus x_4 \oplus x_5)(y_2 \oplus y_3 \oplus y_4 \oplus y_5)) \oplus z_2 \\
f_2 &= (x_1(y_3 \oplus y_4 \oplus y_5)(z_3 \oplus z_4 \oplus z_5) \oplus y_1(x_3 \oplus x_4 \oplus x_5)(z_3 \oplus z_4 \oplus z_5) \\
&\quad \oplus z_1(x_3 \oplus x_4 \oplus x_5)(y_3 \oplus y_4 \oplus y_5) \oplus x_1y_1(z_3 \oplus z_4 \oplus z_5) \oplus x_1z_1(y_3 \oplus y_4 \oplus y_5) \\
&\quad \oplus y_1z_1(x_3 \oplus x_4 \oplus x_5) \oplus x_1y_1z_1) \oplus (x_1(y_3 \oplus y_4 \oplus y_5) \oplus y_1(x_3 \oplus x_4 \oplus x_5) \oplus x_1y_1) \oplus z_3 \\
f_3 &= (x_1y_1z_2 \oplus x_1y_2z_1 \oplus x_2y_1x_1 \oplus x_1y_2z_2 \oplus x_2y_1z_2 \oplus x_2y_2z_1 \oplus x_1y_2z_4 \oplus x_2y_1z_4 \oplus x_1y_4z_2 \\
&\quad \oplus x_2y_4z_1 \oplus x_4y_1z_2 \oplus x_4y_2z_1 \oplus x_1y_2z_5 \oplus x_2y_1z_5 \oplus x_1y_5z_2 \oplus x_2y_5z_1 \oplus x_5y_1z_2 \oplus x_5y_2z_1) \\
&\quad \oplus (x_1y_2 \oplus y_1x_2) \oplus z_4 \\
f_4 &= (x_1y_2z_3 \oplus x_1y_3z_2 \oplus x_2y_1z_3 \oplus x_2y_3z_1 \oplus x_3y_1z_2 \oplus x_3y_2z_1) \oplus 0 \oplus z_5 \\
f_5 &= 0 \oplus 0 \oplus z_1
\end{aligned}
$$